A. B. Veretennikov, An efficient algorithm for three-component key index construction, *Vestn. Udmurtsk. Univ. Mat. Mekh. Komp. Nauki*, 2019, Volume 29, Issue 1, 117–132

DOI: https://doi.org/10.20537/vm190111

КОМПЬЮТЕРНЫЕ НАУКИ 2019. Т. 29. Вып. 1

УДК 519.683.5

© *А. Б. Веретенников*

ЭФФЕКТИВНЫЙ АЛГОРИТМ ДЛЯ СОЗДАНИЯ ПОЛНОТЕКСТОВЫХ ИНДЕКСОВ С ТРЕХКОМПОНЕНТНЫМИ КЛЮЧАМИ

Рассматривается задача полнотекстового поиска с учетом близости в больших текстовых массивах. Пользователь вводит несколько слов в качестве поискового запроса. В результате поиска формируется список документов, содержащих заданные слова. В современных поисковых системах, документы, в которых слова поискового запроса встречаются вблизи, считаются более релевантными. Рассматриваемая задача требует сохранения в индексе информации о каждом вхождении каждого слова в индексируемых текстах. Скорость выполнения поискового запроса зависит от числа вхождений слов запроса в текстах. Следовательно, запросы, включающие часто встречающиеся слова, выполняются существенно медленнее, чем запросы, состоящие из обычных слов. Для каждого слова текста сохраняем в индексах информацию о часто встречающихся словах, которые располагаются в тексте рядом с ним, на расстоянии не более $MaxDistance$. Данный параметр может принимать значения 5, 7 и даже больше. Применение индексов с трехкомпонентными ключами позволяет добиться быстрого выполнения поисковых запросов. Результаты экспериментов поиска, представленные автором ранее, показывают, что среднее время поискового запроса, состоящего из очень часто встречающихся слов, при применении индексов с трехкомпонентными ключами, меньше в 94.7 раза, чем среднее время поиска с использованием обычных инвертированных индексов. В текущей работе рассмотрен новый алгоритм создания индекса с трехкомпонентными ключами. Доказана корректность алгоритма. Представлены результаты экспериментов построения индексов для разных значений параметра $MaxDistance$.

Ключевые слова: полнотекстовый поиск, поисковые системы, инвертированные файлы, дополнительные индексы, поиск с учетом близости слов, индексы с трехкомпонентными ключами.

DOI: 10.20537/vm190111

В данной работе продолжаем исследование [1]. При разработке современных методов полнотекстового поиска считается, что документы, в которых слова запроса располагаются вблизи друг от друга, являются более важными и релевантными [1–4]. Важность учета расстояния между словами для расчета релевантности возрастает с ростом объема текстовой коллекции [3]. Вместе с тем чем больше текстовая коллекция, тем более актуален вопрос обеспечения гарантированного времени обработки поискового запроса, а также возрастает вероятность появления проблем, связанных с производительностью поисковой системы.

Для реализации полнотекстового поиска используются инвертированные индексы [5–8]. Чтобы учитывать расстояние между словами в тексте, для каждого вхождения каждого слова каждого текста требуется сохранять информацию в индексе. Слова в текстах встречаются с разной частотой. Типичное распределение частот встречаемости слов в текстах [9] (закон Ципфа) изображено на рис. 1. Отсортированный список слов, по частоте встречаемости в порядке убывания, задает значения по горизонтальной оси. По вертикали отмечаем суммарное количество вхождений соответствующего слова в текстах.

Скорость выполнения поискового запроса пропорциональна количеству вхождений слов запроса в проиндексированных текстах. Соответственно, запросы, включающие в себя часто встречающиеся слова, могут выполняться существенно дольше (см. рис. 1, слева), чем запросы, состоящие из обычных слов (см. рис. 1, справа). В соответствии с [10] поисковый запрос, который мы рассматриваем как «простой запрос информации» [10], должен выполняться в течение двух секунд или менее. Иначе непрерывность мышления пользователя может быть нарушена, что приведет к существенному снижению эффективности его работы. Если используются обычные инвертированные файлы, то зачастую запросы, включающие в себя часто встречающиеся слова,

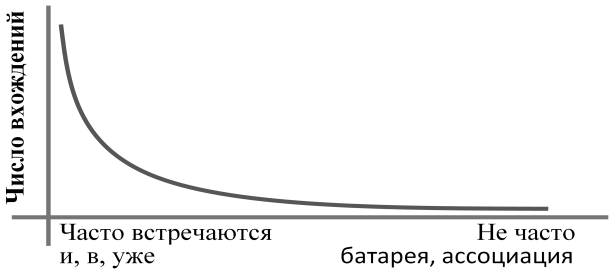

Рис. 1. Пример распределения частот встречаемости слов

выполняются гораздо дольше. Для решения этой проблемы производительности автор использует дополнительные индексы [1, 11, 12]. Преимущества этого подхода, а также рассмотрение других подходов приведены в [1].

Для каждого слова каждого документа можно рассмотреть запись вида (ID, P), где ID — идентификатор документа, например его порядковый номер; P — позиция слова в документе, например порядковый номер слова в документе. Эта запись называется словопозицией. Если есть две словопозиции, A и B, то можно определить, что $A < B$, если выполняется одно из условий: (1) $A.ID < B.ID$ или (2) $A.ID = B.ID$ и $A.P < B.P$.

Среди методов повышения производительности можно отметить следующие:

(1) Методы раннего прерывания поиска (early termination) [13,14] основаны на упорядочении словопозиций в индексе в порядке снижения их важности. Если в процессе чтения списка словопозиций выявляется, что важность текущей словопозиции снизилась до определенного порога, то все оставшиеся словопозиции можно не читать. В такие методы сложно интегрировать учет расстояния между словами, так как может оказаться, что слова запроса располагаются близко друг от друга в документе, который остальными факторами релевантности оценивается как низкорелевантный. При сортировке словопозиций для конкретного ключа, например слова в индексе, нельзя учесть все варианты расположения вблизи этого слова других слов. Более детально см. в [1] комментарии к [2].

(2) Методы создания дополнительных индексов. Разработаны методы для повышения скорости поиска фраз [15], но подобные методы не применимы к поиску с учетом расстояния. По этой причине автор разрабатывает метод [1], который позволяет решить задачу поиска с учетом расстояния.

§ 1. Лемматизация

Морфологический анализатор для каждой словоформы возвращает набор ее базовых форм. Базовые формы также называются леммами, а процесс получения набора базовых форм по словоформе называется лемматизацией. Пусть FL-список — это список всех лемм, упорядоченный в соответствии с убыванием частоты встречаемости леммы в текстах. В [11] введены три типа лемм (в зависимости от частоты встречаемости леммы в текстах).

Стоп-леммы — самые часто встречающиеся леммы, например «и», «в», «или», «уж», «уже», «узкий», «женщина», «война».

Часто используемые леммы — леммы, которые встречаются часто и всегда имеют смысл, например «гора», «полет», «самолет».

Обычные леммы — все остальные леммы, например «орфография», «ассоциация», «рассуждение», «батарея».

Разделение лемм на эти три класса осуществляется путем задания целочисленных параметров $WsCount$ и $FuCount$. В [1] были выбраны значения $WsCount = 700$, $FuCount = 2100$.

Пусть первые $WsCount$ лемм из FL-списка — стоп-леммы. Следующие $FuCount$ лемм в FL-списке — часто используемые леммы. Остальные леммы — обычные.

В некоторых поисковых системах стоп-леммы могут исключаться из поиска и индекса и игнорироваться при поиске. Однако в [1, 15] утверждается, что в определенных случаях стоп-

лемма может иметь особое значение в контексте определенного поискового запроса, и поэтому исключать из рассмотрения стоп-леммы нельзя, приводятся примеры. В нашем подходе обрабатываются все виды лемм.

Следует отметить, что параметр $WsCount = 700$ достаточно большой, и в нашем определении стоп-леммами являются такие леммы, которые очень редко могут быть включены в список стоп-лемм в других подходах, такие как «машина», «женщина» и «работа». Стоп-леммы в нашем понимании — это те леммы, которые встречаются очень часто, и запросы, их включающие, выполняются очень долго.

В качестве $WsCount$ следует выбирать такое значение, чтобы, с одной стороны, запросы не из стоп-лемм выполнялись в течение заданного времени, а с другой — индекс создавался за приемлемое время. Рассмотрим запрос «человек хочет знать» с номерами лемм в FL-списке «человек:67, хотеть:89, знать:82». В условиях [1] данный запрос с использованием дополнительных индексов выполняется 0.031 секунды, а с использованием обычного инвертированного файла — 10.3 секунды. Запрос, включающий менее часто встречающиеся леммы, выполняется быстрее, например «небо солнце корабль вперед», «небо:586, солнце:679, корабль:518, вперед:683», 0.015 с использованием дополнительных индексов и 1.357 с использованием обычного индекса. В данном случае $WsCount = 700$ является пограничным значением таким, что запросы не из стоп-лемм выполняются с использованием обычного инвертированного индекса не более чем за 1–2 секунды.

В [1] также рассмотрена методология поиска, предлагаемая автором, которая включает в себя следующие пункты.

(1) Разделение всех лемм на три класса (в зависимости от частоты встречаемости леммы в текстах): стоп-леммы, часто используемые леммы, обычные леммы.

(2) Применение индексов трехкомпонентных ключей для обработки запросов, состоящих только из стоп-лемм (самый сложный случай с точки зрения производительности).

(3) Применение индексов двухкомпонентных ключей для оптимизации обработки часто используемых лемм в составе запроса.

(4) Включение дополнительной информации в состав словопозиции для однокомпонентных индексов обычных и часто используемых лемм для оптимизации выполнения запросов, включающих в себя как стоп-леммы, так и леммы других типов.

Индексы трехкомпонентных ключей, создание которых рассматривается в данной работе, решают подзадачу обработки запросов, состоящих только из стоп-лемм (пункт (2) методологии). Остальные случаи рассмотрены в [11, 12]. В [1] был рассмотрен вопрос поиска. Сейчас рассмотрим вопрос создания индекса.

§ 2. Алгоритм построения индекса

Индекс трехкомпонентных ключей

Расширенный индекс стоп-лемм или индекс трехкомпонентных ключей [1] — это список словопозиций стоп-леммы f, когда в тексте не более чем на расстоянии $MaxDistance$ от f располагались стоп-леммы s и t. Значения f, s и t являются номерами соответствующих стоп-лемм в FL-списке. $MaxDistance$ — заданный параметр, например 5, 7 или 9.

Если создан расширенный индекс (f, s, t), то данные индексов (s, t, f), (t, f, s) и других вариантов порядка f, s, t, восстанавливаются по индексу (f, s, t). Поэтому расширенный индекс создается только для случая $f \leqslant s \leqslant t$.

В целях записи данных в индексы в многопоточном режиме построим несколько файлов индекса. Файл индекса создается для некоторого подмножества ключей (f, s, t) и содержит в себе расширенные индексы этих ключей.

В каждом файле индекса сохраняются данные определенного подмножества ключей. Разделение множества всех трехкомпонентных ключей на подмножества рассмотрено в [16].

Файл индекса определяется диапазоном допустимых значений первой компоненты ключа. Кроме того, все множество ключей конкретного индекса делим на группы. Группа определяется диапазоном допустимых значений второй компоненты ключа. Разделение множества ключей файла индекса на группы осуществляется в целях оптимизации кэширования при записи индексов. Приведем пример из [16].

Пример конфигурации файлов индекса

Пример 1. При значении $WsCount = 150$ была получена следующая конфигурация файлов индексов:

0: [0, 4] —> [0, 54] [55, 149];
1: [5, 15] —> [5, 32] [33, 60] [61, 104] [105, 149];
2: [16, 52] —> [16, 37] [38, 47] [48, 56] [57, 66] [67, 77] [78, 90] [91, 107] [108, 143] [144, 149];
3: [53, 149] —> [53, 80] [81, 94] [95, 107] [108, 121] [122, 149].

Имеем 4 файла индекса, в первом диапазон значений первой компоненты ключа [0, 4], во втором — [5, 15], в третьем — [16, 52] и в четвертом — [53, 149]. Для каждого файла индекса перечислены группы, заданные диапазоном значений второй компоненты ключа. Рассмотрим для примера, в файле индекса 0, определенным диапазоном [0, 4], первая группа ключей [0, 54], включает в себя ключи:

$$(0, x, y) : 0 \leqslant x \leqslant 54, x \leqslant y; \quad (1, x, y) : 1 \leqslant x \leqslant 54, x \leqslant y; \quad \ldots \quad (4, x, y) : 4 \leqslant x \leqslant 54, x \leqslant y.$$

Данные ключей вида $(5, x, y)$ попадают уже в файл индекса 1.

Построение индексов

Для построения индекса используется подход легко обновляемых индексов [17]. Этот подход позволяет итеративно добавлять в файл индекса данные некоторого подмножества документов. Построение индексов заключается в циклическом повторении двухэтапного процесса.

(1) Последовательное чтение данных из текстовых документов. Сохранение данных в массиве в оперативной памяти, обозначим массив переменной D.
(2) Запись данных из полученного массива D в индексы.

Этап 1. Чтение документов

На первом этапе осуществляется чтение индексируемых документов. Документ — это последовательность слов. Для каждого слова применяется морфологический анализатор, который возвращает набор лемм этого слова $Forms$. Для каждой стоп-леммы x из $Forms$ формируем запись $(ID, P, FL(x))$, где ID — идентификатор документа, P — позиция слова в документе, $FL(x)$ — FL-номер леммы x. Также список $Forms$ обрабатывается для последующего построения других видов индексов.

Примечание. Такая запись требует в среднем 3 байта при кодировании в массиве.

Сформированные записи сохраняются в массиве D в оперативной памяти. Размер массива ограничен. Этап 1 прекращается, когда либо массив заполнен, либо все документы обработаны.

Этап 2. Запись данных из полученного массива в индекс

Исходные данные: массив D записей вида (ID, P, Lem), где ID — идентификатор документа, P — позиция леммы в документе (порядковый номер слова), Lem — FL-номер леммы. Массив D упорядочен по возрастанию значения (ID, P).

Для каждого файла индекса запускается отдельный программный поток. Количество одновременно запущенных программных потоков ограничено. Например, допустим, мы ограничили

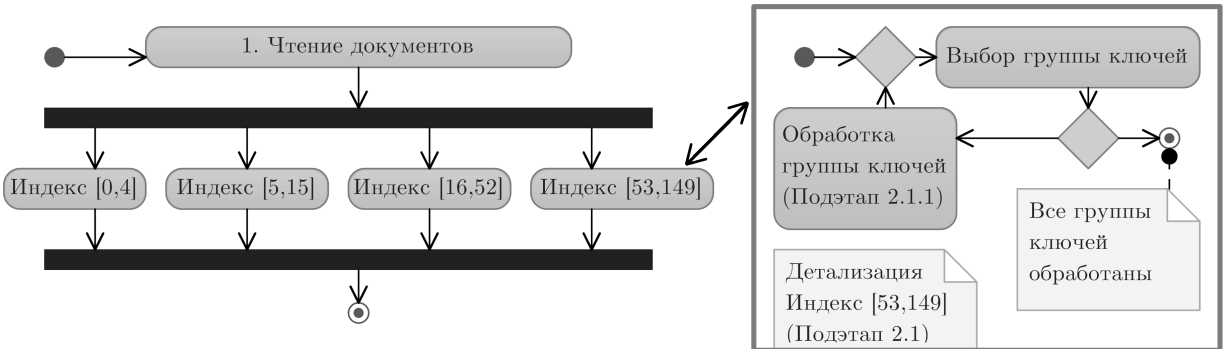

Рис. 2. Двухэтапный процесс создания индекса для примера 1

число одновременно запущенных потоков числом 4, а всего файлов индекса — 7. Вначале будут запущены потоки для первых четырех файлов индекса, а поток для пятого файла индекса будет запущен, когда запись одного из предыдущих файлов индекса будет завершена. Будем обозначать действия, осуществляемые в отдельном программном потоке, как подэтап 2.1.

В одном программном потоке запускается цикл по группам файла индекса. На каждой итерации цикла осуществляется запись данных для ключей (f, s, t), таких, что f входит в допустимый диапазон значений первой компоненты ключа, определенный для индекса, s входит в допустимый диапазон значений второй компоненты ключа, определенный для группы, $f \leqslant s \leqslant t$.

Будем обозначать действия, осуществляемые при обработке одной группы ключей файла индекса, как подэтап 2.1.1. Процесс целиком для примера 1, в предположении, что все потоки на этапе 2 стартуют одновременно, изображен на рис. 2.

Обработка одной группы ключей, подэтап 2.1.1

Все данные массива D читаются последовательно. Сохраняем запись в индексе для каждых трех вхождений лемм f, s, t, если выполнены следующие условия.

Условие 1. Условия для формирования словопозиции для записи в индекс.

(1) $f \leqslant s \leqslant t$.

(2) Ключ (f, s, t) входит в множество ключей текущего файла индекса, то есть f входит в допустимый диапазон значений первой компоненты ключа, определенный для файла индекса.

(3) Лемма s входит в допустимый диапазон значений второй компоненты ключа, определенный для текущей группы.

(4) Расстояния между f и s, а также между f и t по модулю меньше или равны $MaxDistance$ (и не равны нулю).

Алгоритм формирования словопозиций и записи их в индекс для группы ключей

Основная идея алгоритма заключается в следующем. Будем последовательно перебирать записи массива D. Для каждой записи (ID, P) стоп-леммы f будем смотреть, есть ли вблизи нее (то есть на расстоянии не более $MaxDistance$ от f) другие две стоп-леммы s и t. Если такие две стоп-леммы находятся вблизи f и тройка вхождений лемм f, s, t, удовлетворяет вышеуказанным условиям 1, то сохраняем запись в индексе.

Обозначим $[IndexS, IndexE]$ диапазон допустимых значений первой компоненты ключа для текущего индекса.

Обозначим $[GroupS, GroupE]$ диапазон допустимых значений второй компоненты ключа, определенный для текущей группы.

Будем использовать очереди. Структура данных очередь организована в виде односвязного списка и двух указателей: начало очереди и конец очереди. Очередь поддерживает операции:

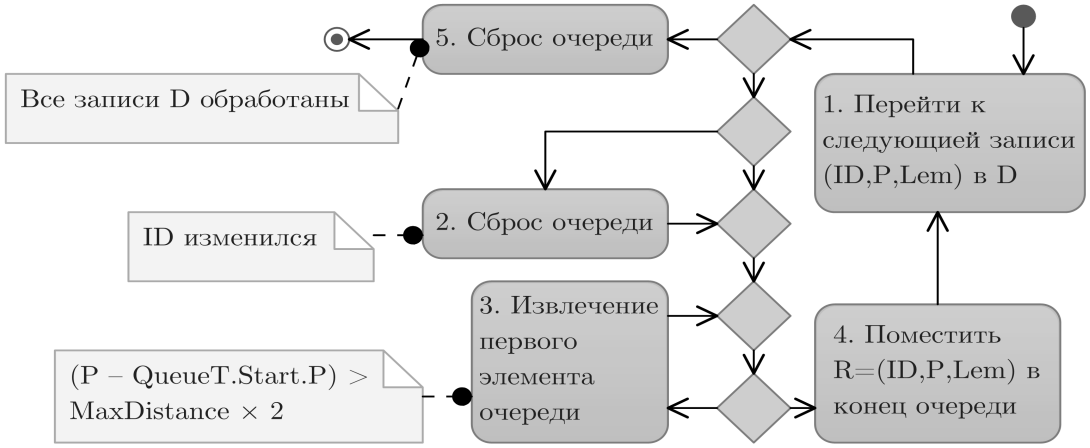

Рис. 3. Упрощенный алгоритм для подэтапа 2.1.1

Рис. 4. Сброс очереди на шагах 2 и 5 на рис. 3

(1) добавить элемент в конец очереди;
(2) удалить элемент с начала очереди (первый элемент);
(3) перебрать элементы очереди, начиная с начала и далее до конца очереди; каждый элемент очереди имеет указатель на следующий элемент очереди; для последнего элемента очереди этот указатель равен $NULL$.

§ 3. Упрощенный алгоритм для подэтапа 2.1.1

Создаем очередь $QueueT$.

В цикле для каждой записи массива D помещаем запись в конец очереди.

Пусть $QueueT.Start$ — начало очереди $QueueT$, $QueueT.End$ — конец очереди $QueueT$.

Для каждого элемента очереди вводим флаг $Processed$ с начальным значением 0, флаг устанавливается при помещении элемента в очередь.

Перед тем как новый элемент (ID, P, Lem) добавляется в очередь, осуществляется проверка. Пока выполняется $(P - QueueT.Start.P) > MaxDistance \times 2$, вызываем процедуру «Извлечение первого элемента очереди».

На рис. 3 изображена UML-диаграмма упрощенного алгоритма для подэтапа 2.1.1. На шагах 2 и 5 осуществляется сброс очереди. Этот процесс отображен на рис. 4 и представляет собой последовательный вызов процедуры «Извлечение первого элемента очереди», пока очередь не станет пуста. Сброс очереди осуществляется при переходе к новому документу, а также при завершении подэтапа 2.1.1. В случае использования узла-развилки на диаграмме комментарием отмечено условие, при котором мы переходим на один из вариантов. Если это условие не выполнено, то осуществляется переход на другой вариант.

Процедура «Извлечение первого элемента очереди»

Формируем и сохраняем словопозиции в индексах (шаги 1, 2, 3, 4 на рис. 5). Для словопозиции нам нужно выбрать три элемента F, S, T очереди $QueueT$, то есть по одному элементу для каждой компоненты ключа (f, s, t) соответственно.

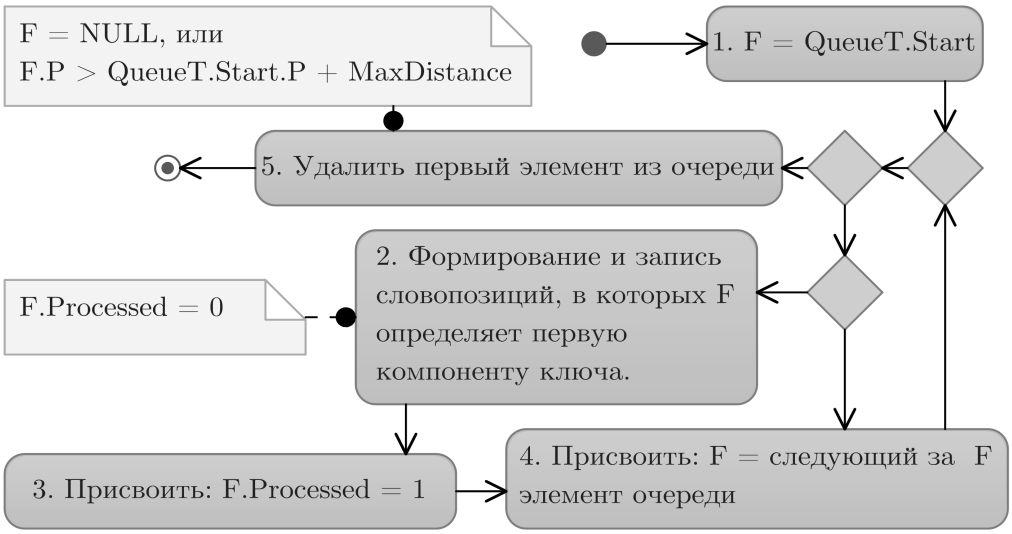

Рис. 5. Процедура «Извлечение первого элемента очереди» для упрощенного алгоритма. Шаг 3 на рис. 3. Шаг 1 на рис. 4

Условие 2. Условия выбора элемента F для первой компоненты ключа.
$F.P \leqslant QueueT.Start.P + MaxDistance$, $F.Processed = 0$, $IndexS \leqslant F.Lem \leqslant IndexE$.

Условие 3. Условия выбора элемента S для второй компоненты ключа.
$|F.P - S.P| \leqslant MaxDistance$, $F.Lem \leqslant S.Lem$, $S.P \neq F.P$, $GroupS \leqslant S.Lem \leqslant GroupE$.

Условие 4. Условия выбора элемента T для третьей компоненты ключа.
$|F.P - T.P| \leqslant MaxDistance$, $S.Lem \leqslant T.Lem$, $T.P \neq F.P$, $T.P \neq S.P$.

Действуем в тройном цикле.

Вначале перебираем возможные элементы F с условиями 2.

Для фиксированного F перебираем возможные элементы S с условиями 3.

Для фиксированных F и S перебираем возможные элементы T с условиями 4.

Выбрав три элемента F, S, T, формируем ключ $(F.Lem, S.Lem, T.Lem)$ и словопозицию $(F.ID, F.P, S.P - F.P, T.P - F.P)$. Словопозицию помещаем в индекс. В словопозицию включены расстояния между компонентами ключа в тексте. При этом расстояние сохраняется со знаком, чтобы можно было определить, до или после леммы $F.Lem$ располагался другой компонент ключа в тексте. Также присваиваем $F.Processed = 1$.

После формирования и записи всех словопозиций удаляем первый элемент очереди.

Процедура изображена на рис. 5.

Переход к новому документу

В очереди находятся записи с одинаковым значением ID.

Если новая запись имеет ID, отличающийся от ID элементов очереди, то последовательно применяется процедура «Извлечение первого элемента очереди», пока очередь не пуста. Этот процесс называется сбросом очереди и отображен на рис. 4.

§ 4. Оптимизированный алгоритм для подэтапа 2.1.1

Создаем три очереди: $QueueF$, $QueueS$, $QueueT$. Последовательно читаем записи массива D. Эти записи упорядочены по возрастанию значения (ID, P).

Пусть текущая прочитанная запись имеет вид $R = (ID, P, Lem)$.

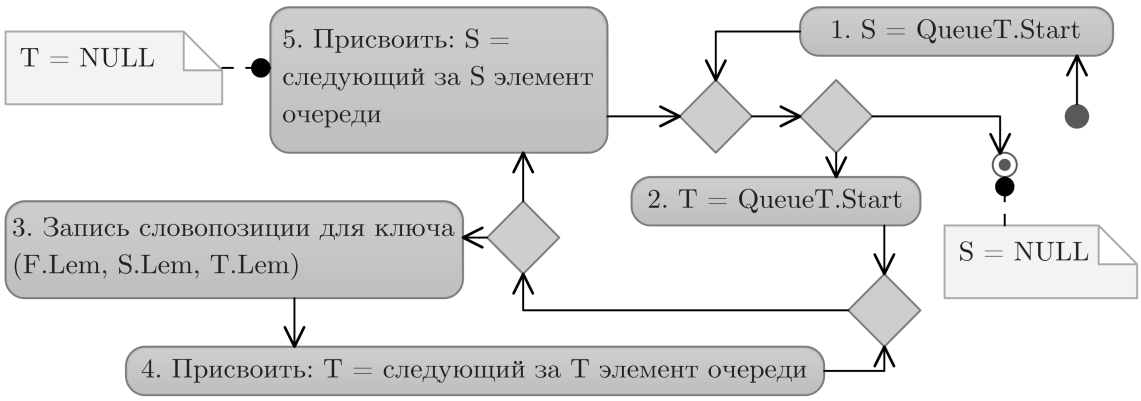

Рис. 6. Шаг 2 процедуры «Извлечение первого элемента очереди» для упрощенного алгоритма, рис. 5

Очередь $QueueF$ предназначена для записей R, соответствующих первой компоненте ключа. Очередь $QueueS$ предназначена для записей R, соответствующих второй компоненте ключа. Очередь $QueueT$ предназначена для записей R, соответствующих третьей компоненте ключа.

Если $R.Lem$ не входит в $[IndexS, IndexE]$, не входит в $[GroupS, GroupE]$, $R.Lem < GroupS$, то такую запись пропускаем (так как $R.Lem$ может быть только третьей компонентой ключа, а в этом случае она должна быть больше или равна $GroupS$). Иначе выполняем следующие действия.

(1) Если $R.Lem$ входит в $[IndexS, IndexE]$, помещаем R в конец очереди $QueueF$.

(2) Если $R.Lem$ входит в $[GroupS, GroupE]$, помещаем R в конец очереди $QueueS$.

(3) Помещаем R в конец очереди $QueueT$.

То есть одна и та же запись R может находиться одновременно в нескольких очередях.

При этом для каждой очереди новый помещаемый элемент больше или равен последнему элементу очереди.

Будем также обеспечивать, чтобы в очередях хранились записи с одинаковым значением поля ID.

Пусть $QueueT.Start$ — первый элемент очереди $QueueT$, $QueueT.End$ — последний добавленный элемент очереди $QueueT$. Будем поддерживать следующее условие:

$(QueueT.End.P - QueueT.Start.P) \leqslant MaxDistance \times 2$.

Перед тем как новый элемент (ID, P, Lem) добавляется в очереди, осуществляется проверка. Пока выполняется условие $(P - QueueT.Start.P) > MaxDistance \times 2$, вызываем процедуру «Извлечение первого элемента очереди».

Процедура «Извлечение первого элемента очереди»

По результатам работы данной процедуры из очереди $QueueT$ будет удален первый, то есть минимальный, элемент. Он также будет удален из очередей $QueueS$, $QueueF$, если он находился в них. Также из очереди $QueueF$ будут удалены все элементы F с условием $F.P \leqslant QueueT.Start.P + MaxDistance$, а все словопозиции вида $(ID, F.P, X, Y)$ для ключей, в которых $F.Lem$ определяет первую компоненту, будут сохранены в индексах (здесь X, Y определяют позиции других компонентов ключа в тексте).

При вызове данной процедуры в очереди $QueueT$ находятся все возможные записи массива D вида (ID, P, Lem), с условием $QueueT.Start.P \leqslant P \leqslant QueueT.Start.P + MaxDistance \times 2$.

Рассмотрим элемент F очереди $QueueF$, удовлетворяющий следующим условиям.

Условие 5. Условия для выбора элемента F для первой компоненты ключа.

(1) $F.P \leqslant QueueT.Start.P + MaxDistance$.

Этот элемент будет соответствовать первой компоненте ключа. Элемент F соответствует вхождению некой леммы f в текстах, которая определит первую компоненту ключа (f, s, t).

В очереди $QueueS$ нужно выбрать элемент S, соответствующий вхождению некой леммы s в текстах, которая определит вторую компоненту ключа (f, s, t). Этот элемент должен удовлетворять следующим условиям.

Условие 6. Условия для выбора элемента S для второй компоненты ключа.

(1) $S.P \neq F.P$ (разные компоненты ключа должны соответствовать словам с разными порядковыми номерами).

(2) $S.P \leqslant F.P + MaxDistance$.

(5) $S.Lem \geqslant F.Lem$ (из условия $f \leqslant s \leqslant t$ для ключа (f, s, t)).

В очереди $QueueT$ нужно выбрать элемент T, соответствующий вхождению некой леммы t в текстах, которая определит третью компоненту ключа (f, s, t). Этот элемент должен удовлетворять следующим условиям.

Условие 7. Условия для выбора элемента T для третьей компоненты ключа.

(1) $T.P \neq F.P$, $T.P \neq S.P$.

(2) $T.P \leqslant F.P + MaxDistance$.

(3) $T.Lem \geqslant S.Lem$, $T.Lem \geqslant F.Lem$.

(4) $T.Lem > S.Lem$ или $\big((T.Lem = S.Lem)$ и $(T.P > S.P)\big)$.

То есть для каждого элемента F очереди $QueueF$, каждого элемента S очереди $QueueS$, каждого элемента T очереди $QueueT$, которые удовлетворяют условиям 5, 6, 7 соответственно, сохраним в индексе $(F.Lem, S.Lem, T.Lem)$ словопозицию $(ID, F.P, S.P - F.P, T.P - F.P)$.

Теорема 1 (о корректности алгоритма). *При вызове процедуры «Извлечение первого элемента очереди», для указанного элемента F из очереди $QueueF$, удовлетворяющего условиям 5, в очереди $QueueT$ находятся все записи $T = (ID, P, Lem)$ массива D, такие, что $|T.P - F.P| \leqslant MaxDistance$.*

Доказательство. Рассмотрим произвольную запись $T = (ID, P, Lem)$ массива D со следующим условием: $|T.P - F.P| \leqslant MaxDistance$.

Если $T.P \geqslant F.P$, то $T.P - F.P \leqslant MaxDistance$, значит, $T.P \leqslant MaxDistance + F.P$.

$F.P \leqslant QueueT.Start.P + MaxDistance$ (условие 5).

$T.P \leqslant MaxDistance + QueueT.Start.P + MaxDistance = QueueT.Start.P + MaxDistance \times 2$.

В очереди $QueueT$ находятся все записи с таким условием, так как процедура «Извлечение первого элемента очереди» вызывается, если для следующей записи N массива D выполняется $(N.P - QueueT.Start.P) > MaxDistance \times 2$ или же все записи массива D обработаны.

Иначе, если $T.P < F.P$, то $F.P - T.P \leqslant MaxDistance$, значит, $T.P \geqslant F.P - MaxDistance$.

Если ранее из $QueueT$ ничего не удалялось, то T находится в $QueueT$.

Иначе пусть Q — предыдущий удаленный элемент из $QueueT$.

$F.P - Q.P > MaxDistance$ (так как все элементы F' с условием $F'.P \leqslant Q.P + MaxDistance$ были обработаны и удалены при предыдущем вызове процедуры из $QueueF$).

$F.P > Q.P + MaxDistance$, следовательно, $T.P > Q.P$.

Значит, T находится в очереди $QueueT$, так как из очереди $QueueT$ ранее удалялись только такие элементы T', для которых выполнялось условие $T'.P \leqslant Q.P$. Вместе с тем, $T.P < F.P$, а F присутствует в $QueueT$, значит T также был добавлен в эту очередь ранее.

Доказательство завершено. □

В оптимизированном алгоритме не нужен флаг $Processed$, так как вместо его присваивания мы удаляем элемент из очереди $QueueF$.

Заметим, что в $QueueS$ находятся те элементы очереди $QueueT$, которые могут соответствовать вхождению второй компоненты ключа. Эти элементы находятся и в $QueueS$, и в $QueueT$. В $QueueS$ они находятся только для оптимизации их перебора.

Последнее условие в 7 требуется для исключения «дубликатов».

Пусть, например, в $QueueT$ есть ровно одна запись F для леммы f. Также для леммы $s \geqslant f$ есть ровно две записи A и B. Если не учитывать условие 6, то можем поместить в индекс словопозиции для ключа (f, s, s): $(ID, F.P, A.P - F.P, B.P - F.P)$ и $(ID, F.P, B.P - F.P, A.P - F.P)$.

Сохранение двух таких записей избыточно.

Чтобы обеспечить то, что в очередях находятся записи с одинаковым значением ID, делается следующее. Так же как и в упрощенном алгоритме, если новая запись R имеет ID, который отличается от ID текущих записей в очередях, выполняем процедуру «Извлечение первого элемента очереди», пока все очереди не станут пусты.

Экспериментальная проверка корректности алгоритма

Теорема 1 обосновывает теоретическую корректность алгоритма. Кроме этого, корректность построенного индекса можно проверить, осуществляя эксперименты поиска. Возьмем некоторый проиндексированный документ. На основании его текста сформируем набор тестовых поисковых запросов. Для проверки корректности индекса нужно осуществить поиск каждого запроса и проверить, что в результатах поиска присутствует запись, соответствующая исходному документу и тому месту в нем, откуда был взят запрос. Методику таких экспериментов и их результаты см. в [1].

§ 5. Ключевые оптимизации и оценка производительности

Периодическое перестроение массива D

Поскольку для каждой группы ключей требуется обработать все записи массива D, объем этого массива существенно влияет на производительность. Для повышения производительности можно разделить процесс записи всех файлов индексов на фазы.

Множество ключей файла индекса ограничено диапазоном изменений первой компоненты ключа, заданного для этого файла индекса. В примере 1 имеем 4 файла индекса: в первом диапазон значений первой компоненты ключа — [0, 4], во втором — [5, 15], в третьем — [16, 52] и в четвертом — [53, 149]. Данные диапазоны идут по возрастанию.

После того как обработаем первый файл индекса, можем исключить из массива D все записи (ID, P, Lem), где $Lem \leqslant 4$, так как для второго файла индекса [5, 15] все компоненты ключа больше или равны 5 (за счет условия $f \leqslant s \leqslant t$ для ключа (f, s, t)). Для третьего индекса [16, 52] все компоненты ключа больше или равны 16.

Если обработаем второй файл индекса, то можем исключить из массива D все записи (ID, P, Lem), где $Lem \leqslant 15$.

Конечно, нет смысла перестраивать массив D после обработки каждого файла индекса, так как в этом случае мы не сможем запускать запись файлов индекса параллельно.

Пример 1 рассмотрен для малого значения параметра $WsCount = 150$. При $WsCount = 700$ имеем 79 файлов индекса. В приведенных экспериментах, результаты которых приведены далее, множество файлов индекса было разбито на 3 группы (15, 23, 41). Соответственно, процесс записи файлов индекса был разделен на 3 фазы. После каждой фазы перестраиваем массив D, удаляя из него записи, которые больше не нужны.

Выравнивание времени записи файла индекса

Важно, чтобы все потоки создания файла индекса выполняли свою работу за примерно одинаковое время. Чем меньше FL-номер леммы, тем чаще она встречается, тем больше записей для ключа, включающего ее. Поэтому для файлов индекса с малым номером диапазон изменения

первой компоненты ключа должен быть меньше. Например, в примере 1 первый файл индекса — [0, 4], а второй — [5, 15].

Но при больших значениях $WsCount$ даже разделения на файлы индекса по первой компоненте может быть недостаточно. В этом случае можно далее делить множество ключей по второй компоненте. Например, для диапазона изменения первой компоненты ключа [5, 15] в примере 1 определены следующие группы ключей, заданные диапазонами изменения второй компоненты ключа: [5, 32], [33, 60], [61, 104], [105, 149]. Можно создать два файла индекса вместо одного, для обоих диапазон изменения первой компоненты ключа — [5, 15], но для первого мы используем группы [5, 32], [33, 60], а для второго, скажем, [61, 104], [105, 149].

Коэффициент утилизации

Количество одновременно работающих потоков записи файлов индекса ограничено. Для каждого ключа необходимо выделить место в кэше [17]. Обычно число файлов индекса таково, что все потоки не могут быть сразу запущены.

Ресурсы распределены оптимально, если в каждый момент времени запущено максимальное число потоков. Как только один поток завершает создание своего файла индекса, стартует следующий поток. При этом нужно стремиться к тому, чтобы при окончании записи файлов индекса было запущено максимальное число потоков и все запущенные потоки завершали бы свою работу примерно в одно время.

В переменной $RefCount$ храним число запущенных потоков. Когда поток стартует или прекращает свою работу, меняем счетчик загруженных потоков $RefCount$. Замеряем также время $Delta$, сколько прошло с момента последнего изменения счетчика, формируем запись $(RefCount, Delta)$ и помещаем ее в список.

Таким образом, есть набор n записей вида $(RefCount_i, Delta_i)$, $1 \leqslant i \leqslant n$, каждая из которых соответствует некоторому промежутку времени длиной $Delta_i$ секунд, где n — количество запусков и остановок потоков. На протяжении промежутка времени $Delta_i$ было запущено $RefCount_i$ потоков. Переменная $MaxRefCount$ хранит в себе максимальное когда-либо число одновременно работающих потоков, $MaxRefCount = \max\limits_{i=1}^{n} RefCount_i$.

Весь промежуток времени записи файлов индекса представляет собой разбиение на n указанных промежутков, определяемых списком записей $(RefCount_i, Delta_i)$.

Вычисляем $TotalDelta = \sum\limits_{i=1}^{n} Delta_i$ — время записи файлов индекса.

Вычисляем: $U = \left(\sum\limits_{i=1}^{n} (RefCount_i \times Delta_i)\right) / \left(\sum\limits_{i=1}^{n} (MaxRefCount \times Delta_i)\right)$.

Это значение называем коэффициентом утилизации. Если коэффициент утилизации равен 1, значит, когда осуществлялась запись индексов, всегда, в каждый момент времени, было запущено максимальное число потоков, и они одновременно завершили свою работу. Теоретически этого можно достичь, например, если каждый файл индекса записывался бы за одно и то же время и число файлов индекса нацело делилось бы на $MaxRefCount$. Также $U = 1$, если все потоки индексации можно запустить сразу, и каждый из них работает за одно и то же время. Если U близок к 1, значит вычислительные ресурсы используются эффективно.

В проведенных экспериментах коэффициент утилизации U был не ниже 0.8.

Коэффициент максимальной загрузки M определяет, в какую долю времени работало максимальное число потоков. $M = \left(\sum\limits_{i=1}^{n} (eq(RefCount_i, MaxRefCount) \times Delta_i)\right) / TotalDelta$, где $eq(a, b) = \begin{cases} 1, & \text{если } a = b, \\ 0, & \text{если } a \neq b. \end{cases}$ M принимал значение от 0.55 до 0.8.

§ 6. Результаты экспериментов

Эксперименты проведены в следующем окружении.

Intel Xeon X5650 2,67 GHZ (2 процессора, 6 ядер каждый), 48 Гб. RAM.

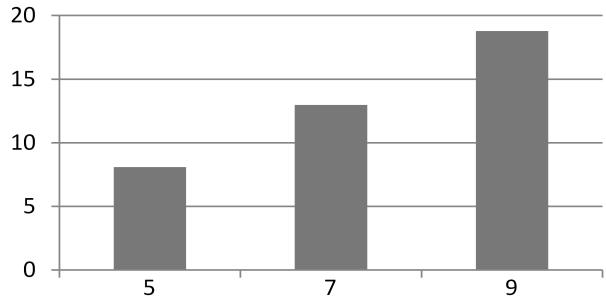

Рис. 7. Время записи индексов трехкомпонентных ключей (часы) для $MaxDistance = 5, 7, 9$

Windows Server 2008 R2 Enterprise, x64 bit.

Индексы создавались на одном жестком диске HGST HUS726060AL, 6 Tb, SATA.

Для экспериментов использовались те же тексты, что и в [1].

Проиндексировано 195 тыс. файлов, объем текста 71,5 Гб, файлы представляли собой обычный текст, однобайтовая кодировка, по стилю — художественная литература, в основном русский язык. Запросы также состоят из слов русского языка.

$WsCount = 700$, $FuCount = 2100$. Были созданы три индекса для $MaxDistance = 5, 7$ и 9, назовем их Idx1, Idx2, Idx3.

Суммарные размеры, включая все индексы: Idx1: 746 Гб., Idx2: 1.23 Тб., Idx3: 1.88 Тб.

Время создания, включая все индексы:

Idx1: 14 ч 43 мин, Idx2: 19 ч 31 мин, Idx3: 26 ч 49 мин.

Размеры индексов трехкомпонентных ключей: Idx1: 425 Гб., Idx2: 883 Гб., Idx3: 1.45 Тб.

Время записи индексов трехкомпонентных ключей (см. рис. 7).

Idx1: 8 ч 06 мин, Idx2: 12 ч 39 мин, Idx3: 18 ч 47 мин.

§ 7. Дополнительные вопросы

В приведенных выше экспериментах данные хранились в наборе файлов индекса. Использовалась файловая система NTFS. Для изучения, насколько особенности файловой системы могут влиять на производительность, был проведен дополнительный эксперимент. Был создан индекс с настройкой $MaxDistance = 5$ с сохранением данных напрямую на логический том.

Поскольку требуется создавать набор файлов, была создана примитивная файловая система. Структура файловой системы следующая. Каждый файл сохраняется в наборе больших блоков. Размер блока 64 Мб. Отслеживается объем занятого места на логическом томе. При необходимости увеличить размер файла, в конец занятой области добавляется новый блок и добавляется в список блоков файла. В этих условиях индекс был создан за 13 ч 53 мин, это немного меньше, чем 14 ч 43 мин, время, за которое Idx1 был создан ранее. Среднее время поиска не изменилось.

Размеры индексов существенно превышают размеры обычных индексов. Вместе с тем, с учетом существенного ускорения поиска, предлагаемый метод позволит существенно сократить стоимость поисковой системы во многих случаях. Обеспечить дополнительный объем дискового пространства часто гораздо легче, чем добавить несколько новых серверов. Возникает также вопрос, не получится ли каким-либо образом сжать индекс.

В рамках предварительного анализа были осуществлены эксперименты сжатия данных индекса Zip-архиватором. Осуществлены попытки сжатия как отдельных списков словопозиций после их извлечения из индекса в отдельные файлы, так и всего индекса. Сжатые данные занимали примерно 70 % от исходных.

Рассмотрим далее вопросы, связанные с релевантностью.

Результат запроса Q, состоящего из n слов, — это набор записей. Каждая запись содержит ID — идентификатор документа и список позиций слов запроса в тексте $X = (X_1, \ldots, X_n)$.

В [2] релевантность определяется функцией вида $S = \alpha \cdot SR + \beta \cdot IR + \gamma \cdot TP$, где SR — статический ранг документа ID, не зависящий от поискового запроса, например PageRank. IR

позволяет учитывать слова запроса, например BM25. TP учитывает расстояние между словами запроса в тексте документа для конкретного результата поиска. Значения SR, IR, TP нормализованные, то есть это числа в диапазоне от 0 до 1, α, β, γ — параметры.

Часто значение TP определяется числом или числами, обратно пропорциональными квадрату расстояния между словами. В случае [2] запрос состоит из двух слов и $TP(A,B) = 1/|A-B|^2$, где A и B — позиции слов запроса в тексте. В [18] также для оценки TP используются значения, обратно пропорциональные квадрату расстояния между словами запроса в документе, эта информация комбинируется с расчетом BM25 для расчета релевантности.

Для случая, если запрос состоит из нескольких слов, предлагаем рассмотреть функцию $TP(X) = 1\Big/\Big(|A(X) - B(X)| - (n-2)\Big)^2$, где $A(X) = \min\limits_{1 \leqslant i \leqslant n} X_i$, $B(X) = \max\limits_{1 \leqslant i \leqslant n} X_i$.

Проведенные автором эксперименты показали, что для запросов, состоящих из часто встречающихся слов, BM25 дает похожие результаты для большого числа документов (поиск с использованием обычного инвертированного индекса). Введем параметр $RankBorder = 0.9$. Чем чаще слова запроса встречаются в текстах, тем больше записей в результатах поиска имеют $BM25 \geqslant RankBorder$; для запросов, состоящих из самых часто встречаемых слов, число таких записей достигает десятков тысяч (средняя длина документа в рассматриваемой текстовой коллекции — около 54 тыс. слов). Автор предполагает, что для часто встречающихся слов, которые встречаются практически в каждом документе, критерии релевантности, основанные на информации о том, входит ли слово в документ и с какой частотой, такие как BM25, работают плохо, и в этом случае основной критерий для определения релевантности — значение TP.

При поиске в расширенном индексе обрабатываются только такие случаи, когда слова запроса располагаются в тексте на расстоянии не более $MaxDistance$. Требуется выбрать достаточно большое значение параметра, чтобы все вхождения запроса в текстах с высоким значением TP были обработаны. В случае когда TP определяется в виде, обратно пропорциональном квадрату расстояния между словами запроса в документе, это можно сделать. Например, если $MaxDistance = 9$, то для любого запроса длины $\leqslant 7$, с условием $|A(X) - B(X)| > 9$, имеем $TP \leqslant 1/25 = 0.04$. Рассмотрим запрос из 7-ми слов. Если слова запроса встречаются в тексте в виде точной фразы, то $|A(X) - B(X)| = 6$ и $TP = 1$. Чем больше «лишних» слов между словами запроса в тексте, тем меньше TP. Если $|A(X) - B(X)| = 10$, то $TP = 1/(10-5)^2 = 0.04$. Таким образом, все вхождения запроса длины $\leqslant 7$ в текстах с высоким $TP > 0.04$ будут найдены с использованием расширенного индекса. Запросы, состоящие из большого числа слов, следует разделять на части [1].

Что делать, если имеем большое значение SR, а значение TP мало и подобные вхождения запроса в текстах могут быть пропущены при поиске в расширенном индексе? Можно поставить вопрос о том, релевантны ли вообще такие результаты; скорее всего, их можно пропустить. Также можно выполнить дополнительно запрос без учета расстояния [1], он выполняется существенно быстрее, так как используется индекс, в котором для каждого слова каждого документа сохраняется информация только о первом вхождении этого слова в документе.

§ 8. Заключение

Рассмотрен алгоритм создания индекса трехкомпонентных ключей. Приведены результаты экспериментов создания индекса. Достигнуты следующие цели.

Показано, что индексы трехкомпонентных ключей могут быть созданы для относительно большого значения параметра $MaxDistance$ (созданы индексы для значений 5, 7, 9).

Выявлено, что при увеличении параметра $MaxDistance$ размеры индексов трехкомпонентных ключей и времена их создания существенно возрастают.

Выявлено, что размеры созданных индексов существенно превышают размеры обычных инвертированных индексов, но это не являются критичным с учетом размера современных носителей информации.

Выявлено, что текущий алгоритм является требовательным к ресурсам процессора, и скорее ресурсы процессора, чем скорость ввода-вывода, являются фактором, ограничивающим скорость

создания индекса.

Доказана корректность алгоритма построения индекса.

Введен коэффициент утилизации как критерий эффективного использования вычислительных ресурсов. Зафиксированное в процессе экспериментов значение коэффициента утилизации показывает, что ресурсы использовались эффективно.

В будущем представляет интерес оптимизация рассмотренного алгоритма в целях сокращения времени построения индекса. Представляет интерес вопрос определения оптимальных значений параметров $WsCount$ и $FuCount$ в части обеспечения минимального времени создания индекса при сохранении максимальной скорости поиска в построенном индексе.

Финансирование. Работа выполнена при финансовой поддержке постановления № 211 Правительства Российской Федерации, контракт № 02.A03.21.0006.

СПИСОК ЛИТЕРАТУРЫ

1. Веретенников А.Б. Применение трехкомпонентных ключей для полнотекстового поиска с учетом расстояния с гарантированным временем отклика // Вестник ЮУрГУ. Сер. Вычислительная математика и информатика. 2018. Т. 7. № 1. С. 60–77. https://doi.org/10.14529/cmse180105
2. Yan H., Shi S., Zhang F., Suel T., Wen J.-R. Efficient term proximity search with term-pair indexes // Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM'10). Toronto, Canada, 2010. P. 1229–1238. https://doi.org/10.1145/1871437.1871593
3. Buttcher S., Clarke C., Lushman B. Term proximity scoring for ad-hoc retrieval on very large text collections // Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'06). Seattle, USA, 2006. P. 621–622. https://doi.org/10.1145/1148170.1148285
4. Rasolofo Y., Savoy J. Term proximity scoring for keyword-based retrieval systems // European Conference on Information Retrieval (ECIR'2003): Advances in Information Retrieval. 2003. P. 207–218. https://doi.org/10.1007/3-540-36618-0_15
5. Zobel J., Moffat A. Inverted files for text search engines // ACM Computing Surveys. 2006. Vol. 38. No. 2. Article 6. https://doi.org/10.1145/1132956.1132959
6. Tomasic A., Garcia-Molina H., Shoens K. Incremental updates of inverted lists for text document retrieval // Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94). Minneapolis, Minnesota, USA, 1994. P. 289–300. https://doi.org/10.1145/191839.191896
7. Brown E.W., Callan J.P., Croft W.B. Fast incremental indexing for full-text information retrieval // Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94). Santiago de Chile, Chile, 1994. P. 192–202.
8. Luk R.W.P. Scalable, statistical storage allocation for extensible inverted file construction // Journal of Systems and Software. 2011. Vol. 84. No. 7. P. 1082–1088. https://doi.org/10.1016/j.jss.2011.01.049
9. Zipf G. Relative frequency as a determinant of phonetic change // Harvard Studies in Classical Philology. 1929. Vol. 40. P. 1–95. https://doi.org/10.2307/310585
10. Miller R.B. Response time in man–computer conversational transactions // Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, part I (AFIPS'68). San Francisco, California, 1968. P. 267–277. https://doi.org/10.1145/1476589.1476628
11. Веретенников А.Б. Использование дополнительных индексов для более быстрого полнотекстового поиска фраз, включающих часто встречающиеся слова // Системы управления и информационные технологии. 2013. Т. 52. № 2. С. 61–66.
12. Веретенников А.Б. Эффективный полнотекстовый поиск с использованием дополнительных индексов часто встречающихся слов // Системы управления и информационные технологии. 2016. Т. 66. № 4. С. 52–60.
13. Anh V.N., de Kretser O., Moffat A. Vector-space ranking with effective early termination // Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'01). New Orleans, Louisiana, USA, 2001. P. 35–42. https://doi.org/10.1145/383952.383957
14. Garcia S., Williams H.E., Cannane A. Access-ordered indexes // Proceedings of the 27th Australasian Conference on Computer Science (ACSC'04). Dunedin, New Zealand, 2004. P. 7–14.

15. Williams H.E., Zobel J., Bahle D. Fast phrase querying with combined indexes // ACM Transactions on Information Systems (TOIS). 2004. Vol. 22. No. 4. P. 573–594. https://doi.org/10.1145/1028099.1028102
16. Веретенников А.Б. Эффективный полнотекстовый поиск с учетом близости слов при помощи трехкомпонентных ключей // Системы управления и информационные технологии. 2017. Т. 69. № 3. С. 25–32.
17. Веретенников А.Б. О структуре легко обновляемых полнотекстовых индексов // Современные проблемы математики и ее приложений. Труды Международной (48-й Всероссийской) молодежной школы-конференции. 2017. С. 30–41. http://ceur-ws.org/Vol-1894/
18. Lu X., Moffat A., Culpepper J.S. Efficient and effective higher order proximity modeling // Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval (ICTIR'16). Newark, Delaware, USA, 2016. P. 21–30. https://doi.org/10.1145/2970398.2970404

Поступила в редакцию 01.07.2018

Веретенников Александр Борисович, к. ф.-м. н., доцент, кафедра вычислительной математики и компьютерных наук, Уральский федеральный университет, 620083, Россия, г. Екатеринбург, пр. Ленина, 51.
E-mail: alexander@veretennikov.ru

A. B. Veretennikov

An efficient algorithm for three-component key index construction

Citation: *Vestnik Udmurtskogo Universiteta. Matematika. Mekhanika. Komp'yuternye Nauki*, 2019, vol. 29, issue 1, pp. 117–132 (in Russian).

Keywords: full-text search, search engines, inverted files, additional indexes, proximity search, three-component key indexes.

MSC2010: 68P20, 68P10

DOI: 10.20537/vm190111

Proximity full-text searches in large text arrays are considered. A search query consists of several words. The search result is a list of documents containing these words. In a modern search system, documents that contain search query words that are near each other are more relevant than other documents. To solve this task, for each word in each indexed document, we need to store a record in the index. In this case, the query search time is proportional to the number of occurrences of the queried words in the indexed documents. Consequently, it is common for search systems to evaluate queries that contain frequently occurring words much more slowly than queries that contain less frequently occurring, ordinary words. For each word in the text, we use additional indexes to store information about nearby words at distances from the given word of less than or equal to $MaxDistance$, which is a parameter. This parameter can take a value of 5, 7, or even more. Three-component key indexes can be created for faster query execution. Previously, we presented the results of experiments showing that, when queries contain very frequently occurring words, the average time of the query execution with three-component key indexes is 94.7 times less than that required when using ordinary inverted indexes. In the current work, we describe a new three-component key index building algorithm. We prove the correctness of the algorithm. We present the results of experiments of the index creation depending on the value of $MaxDistance$.

Funding. The work was supported by Act 211 Government of the Russian Federation, contract № 02.A03.21.0006

REFERENCES

1. Veretennikov A.B. Proximity full-text search with response time guarantee by means of three component keys, *Bulletin of the South Ural State University. Ser. Computational Mathematics and Software Engineering*, 2018, vol. 7, no. 1, pp. 60–77 (in Russian). https://doi.org/10.14529/cmse180105

2. Yan H., Shi S., Zhang F., Suel T., Wen J.-R. Efficient term proximity search with term-pair indexes, *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM'10)*, Toronto, Canada, 2010, pp. 1229–1238. https://doi.org/10.1145/1871437.1871593
3. Buttcher S., Clarke C., Lushman B. Term proximity scoring for ad-hoc retrieval on very large text collections, *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'06)*, Seattle, USA, 2006, pp. 621–622. https://doi.org/10.1145/1148170.1148285
4. Rasolofo Y., Savoy J. Term proximity scoring for keyword-based retrieval systems, *European Conference on Information Retrieval (ECIR'2003): Advances in Information Retrieval*, 2003, pp. 207–218. https://doi.org/10.1007/3-540-36618-0_15
5. Zobel J., Moffat A. Inverted files for text search engines, *ACM Computing Surveys*, 2006, vol. 38, no. 2, article 6. https://doi.org/10.1145/1132956.1132959
6. Tomasic A., Garcia-Molina H., Shoens K. Incremental updates of inverted lists for text document retrieval, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, Minneapolis, Minnesota, USA, 1994, pp. 289–300. https://doi.org/10.1145/191839.191896
7. Brown E.W., Callan J.P., Croft W.B. Fast incremental indexing for full-text information retrieval, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago de Chile, Chile, 1994, pp. 192–202.
8. Luk R.W.P. Scalable, statistical storage allocation for extensible inverted file construction, *Journal of Systems and Software*, 2011, vol. 84, no. 7, pp. 1082–1088. https://doi.org/10.1016/j.jss.2011.01.049
9. Zipf G. Relative frequency as a determinant of phonetic change, *Harvard Studies in Classical Philology*, 1929, vol. 40, pp. 1–95. https://doi.org/10.2307/310585
10. Miller R.B. Response time in man–computer conversational transactions, *Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, part I (AFIPS'68)*, San Francisco, California, 1968, pp. 267–277. https://doi.org/10.1145/1476589.1476628
11. Veretennikov A.B. Using additional indexes for fast full-text searching phrases that contains frequently used words, *Sistemy Upravleniya i Informatsionnye Tekhnologii*, 2013, vol. 52, no. 2, pp. 61–66 (in Russian).
12. Veretennikov A.B. Efficient full-text search by means of additional indexes of frequently used words, *Sistemy Upravleniya i Informatsionnye Tekhnologii*, 2016, vol. 66, no. 4, pp. 52–60 (in Russian).
13. Anh V.N., de Kretser O., Moffat A. Vector-space ranking with effective early termination, *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'01)*, New Orleans, Louisiana, USA, 2001, pp. 35–42. https://doi.org/10.1145/383952.383957
14. Garcia S., Williams H.E., Cannane A. Access-ordered indexes, *Proceedings of the 27th Australasian Conference on Computer Science (ACSC'04)*, Dunedin, New Zealand, 2004, pp. 7–14.
15. Williams H.E., Zobel J., Bahle D. Fast phrase querying with combined indexes, *ACM Transactions on Information Systems (TOIS)*, 2004, vol. 22, no. 4, pp. 573–594. https://doi.org/10.1145/1028099.1028102
16. Veretennikov A.B. Efficient full-text proximity search by means of three component keys, *Sistemy Upravleniya i Informatsionnye Tekhnologii*, 2017, vol. 69, no. 3, pp. 25–32 (in Russian).
17. Veretennikov A.B. About a structure of easy updatable full-text indexes, *Proceedings of the International Youth School–Conference "SoProMat-2017"*, Yekaterinburg, Russia, 2017, pp. 30–41 (in Russian). http://ceur-ws.org/Vol-1894/
18. Lu X., Moffat A., Culpepper J.S. Efficient and effective higher order proximity modeling, *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval (ICTIR'16)*, Newark, Delaware, USA, 2016, pp. 21–30. https://doi.org/10.1145/2970398.2970404

Received 01.07.2018

Veretennikov Aleksandr Borisovich, Candidate of Physics and Mathematics, Associate Professor, Department of Calculation Mathematics and Computer Science, Ural Federal University, pr. Lenina, 51, Yekaterinburg, 620083, Russia.
E-mail: alexander@veretennikov.ru